**Module 3: Linked Structures and Process Control**

Up to this point, we have worked with arrays and strings—data structures with fixed sizes and direct indexing. But real-world problems rarely stay fixed. The number of items may grow, shrink, or change unpredictably. Arrays struggle in such situations, because they require continuous memory and often involve shifting elements when inserting or deleting.

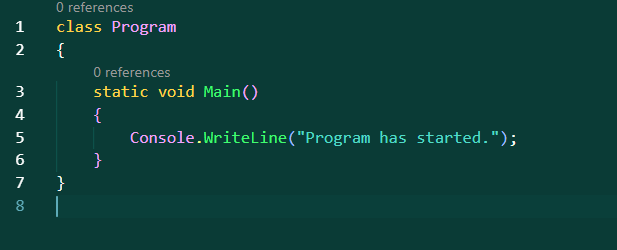
This is where linked structures come in. Unlike arrays, linked structures do not depend on fixed memory blocks. Each element (called a node) connects to the next one through a reference, creating a chain of data. This allows data to grow and shrink freely as needed. Along with stacks and queues, linked structures also determine the order of execution in programs. They are the foundation of how operating systems handle tasks, how functions call each other, and how data is processed in real time.

By the end of this module, you should understand how data moves through linked lists, stacks, and queues, and why these structures are essential when we need flexibility and order in our programs.

**Core Components**

**1. Class Program**

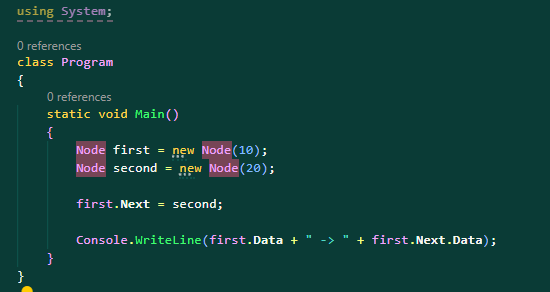
This is always the entry point of a C# console application.  
It contains the Main() method where we test our data structures.

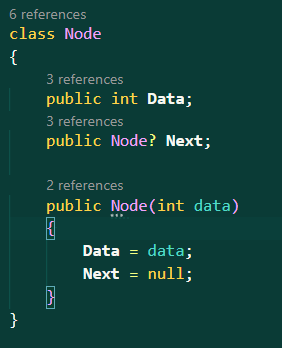


**2. Linked Lists**

A linked list is like a chain of connected boxes. Each box, called a node, stores two things: the actual data and a reference to the next node. This design makes linked lists more flexible than arrays, because new nodes can be added or removed without rearranging the entire structure.

In C#, we usually define a node with a small class. For example:



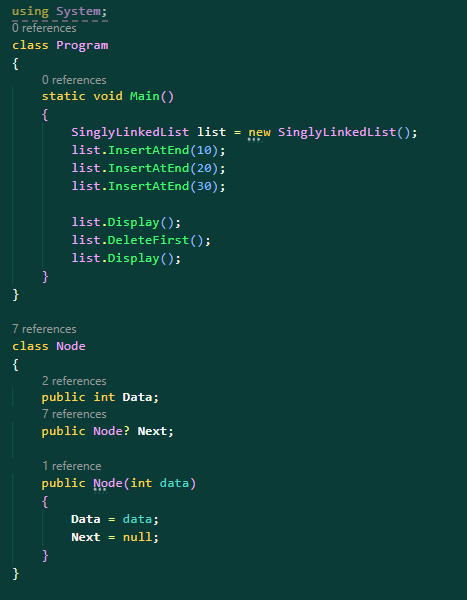


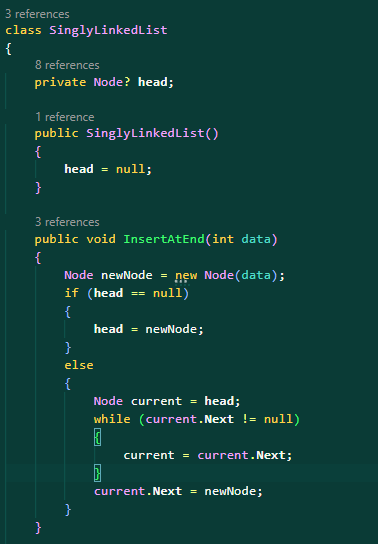
* class Node: This defines a class named Node, which is typically used to represent an element in a linked list.
* public int Data;: This is a public field that stores an integer value for the node. It holds the data for this node.
* public Node? Next;: This is a public field that holds a reference to the next node in the list. The ? means it can be null (i.e., there may not be a next node).
* public Node(int data): This is a constructor. When you create a new Node, you pass an integer value (data), which gets assigned to the Data field.
* Data = data;: Sets the node’s data to the value provided.
* Next = null;: Initializes the Next reference to null, meaning this node does not point to any other node yet.

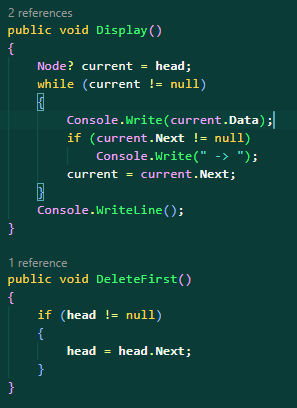
This simple class creates a node that stores a number and a link to another node. Notice that Next is of type Node, allowing us to connect one node to another.

**SinglyLinkedList**

To manage a list of nodes, we create a linked list class that keeps track of the first node, also called the head. For example:



Here, InsertAtEnd adds a new node at the end of the list. If the list is empty, the new node becomes the head. Otherwise, the method walks through the list until it reaches the last node, then attaches the new one.



Display walks through the list, starting from the head, printing each node’s data, and following the chain until it reaches the end.

DeleteFirst shows how easy it is to remove the first node. Instead of shifting elements as we do in arrays, we simply move the head reference to the next node.

This example demonstrates the power of linked lists: they allow dynamic growth and easy reorganization. However, they are slower than arrays when you need direct access, because finding a specific element requires starting at the head and moving step by step.

**Stacks**

A stack is another structure, but instead of nodes, it is defined by its behavior. A stack works on the principle of Last In, First Out (LIFO). Imagine stacking books: the last book you put on top is the first one you remove.

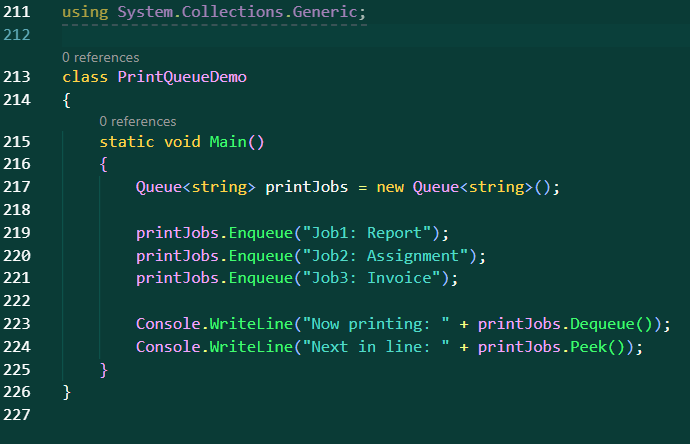
Here’s a simple example: 

In this example, we use two stacks to simulate undo and redo actions. Every time something is typed, it is pushed onto the undo stack. When undoing, we pop from the undo stack and push onto the redo stack. When redoing, we pop from redo and push back onto undo.

This demonstrates how editors and IDEs keep track of changes. Each operation is stored, and the stack ensures that the last change is the first one to be undone. The efficiency of stack operations is very high, with push, pop, and peek all taking constant time O(1).

**Queues**

A queue works in the opposite way: First In, First Out (FIFO). Think of a line at a store. The first person to arrive is the first person to be served.

C# has a built-in Queue<T> class. Consider this example of a print queue:

Here, we add print jobs with Enqueue. The first job entered is the first one removed with Dequeue. If we want to see what is next without removing it, we use Peek.

This is how real-world systems like printers, operating systems, and customer service lines manage tasks. Operations in queues are also efficient, with enqueue and dequeue being constant time O(1), though traversing the entire queue takes O(n).

**CORE COMPONENTS RECAP**

**1. class Program**

The entry point of execution. Contains the Main() method where we test our data structures.

**2. class Node**

Defines the structure of a single node in a linked list. Holds Data and a reference to the next node (Next).

**3. class SinglyLinkedList**

Manages a chain of nodes. Includes:

* InsertAtEnd(int data) → adds a new node at the end.
* DeleteFirst() → removes the first node.
* Display() → prints all nodes.

**4. Stack<T> (from System.Collections.Generic)**

A built-in C# generic class that works on **Last In, First Out (LIFO)** principle.

* Push(T item) → adds an element to the top.
* Pop() → removes and returns the top element.
* Peek() → returns the top element without removing it.

**5. Queue<T> (from System.Collections.Generic)**

A built-in C# generic class that works on **First In, First Out (FIFO)** principle.

* Enqueue(T item) → adds an element to the back.
* Dequeue() → removes and returns the front element.
* Peek() → returns the front element without removing it.